Cosc460
Honours Research Project
Department Of Computer Science
University Of Canterbury

G P S S / 750

SIMULATION SYSTEM

Supervisor : Dr W. Kreutzer

Ting Chin Kee

13. 10. 82

CONTENTS

PREFACE

In the field of software engineering, it is now recognized that proper language design can greatly improve the reliability, clarity, ease of implementation and maintenance of software written in that language. Two valuable features of languages designed for the implementation of reliable software are 'strong typing' and structured control statements (e.g. if-then-else and while-do). Basically, 'strong typing' means that every variable must be declared to be of a specific type before it is used, and only expressions of that type may be used to assign new values to the variable.

However, despite evidence that shows the usefulness of strongly-typed languages, most simulation languages do not have this feature. At present, the best known simulation language with strong-typing is SIMULA and it is implemented on only a few machines since a SIMULA compiler is relatively complex and expensive to implement.

SIMPAS is another strongly-typed simulation language developed at the University of Wisconsin in early 1981. Like many other new simulation language, it is not widely known and used at present, mainly because users are generally reluctant to learn a new language just for the purpose of writing a simulation.

The easiest way to introduce new software features into the user community is via the implementation of a suitably modified (with regard to data declarations and control flow structuring facilities) subset of a widely used simulation language. Such a language would be more readily accepted by the user community, especially among the users of the original language because those users will not be required to learn a completely new language.

This report introduces and discusses the design and implementation of a modified subset of the widely used General Purpose Simulation System (GPSS) on the PRIME/750 computer system at the University of Canterbury computer centre. It is assumed that the reader is familiar with the basic concepts of discrete event simulation systems.

The first section is an introduction to the design of the new system, introducing the basic concepts, data structures and the internal organization of the system.

The second section presents the language specifications for the new system.

The third section discusses the implementation of the system, the organization and structure of the software written for the system, and a brief discussion on the difficulties encountered during the implementation process.

ACKNOWLEDGEMENTS

My special thanks go to Dr W. Kreutzer for his invaluable guidance and support in this project.

1 GPSS/750 SIMULATION SYSTEM

1.1 Introduction

GPSS/750 is designed to be a subset of the widely used General Purpose Simulation System which was developed by Geoffrey Gordon and first released by IBM in 1962. It is implemented as a PASCAL based preprocessor system which accepts a GPSS/750 program as input and produces a standard PASCAL program and a program listing as output. The preprocessor itself is written in standard PASCAL so that it may be used on any system which supports standard PASCAL.

The GPSS language has a few undesirable properties which make the design and development of reliable, well structured models a very difficult task. Specifically :

1. It is inefficient in terms of time and space.
2. It has rather poor flow control and modularization structures.
3. It does not provide any type checking facility.
4. Data input is not flexible; it is difficult to use externally generated or externally supplied data.
5. The programmer has little control over the output.

Using PASCAL as the host language for GPSS/750 system provides the programmer with a lot more programming flexibility and safety to overcome most of the undesirable properties of GPSS :

1. PASCAL procedures or other constructs may be included in a GPSS/750 program to manipulate the status and attributes of any of the entities used in the simulation model before or after a simulation run. This provides the user with a powerful tool to override the normal GPSS/750 sequence of operations in order to model special conditions.
2. PASCAL input/output systems make it easier for reading externally supplied data into the system.
3. PASCAL structured control statements may be embedded in the program to enhance flow control and modularity of the program.

4. The user may write his own report generator in PASCAL to produce his own report to suit his needs.
5. The type checking inherent in GPSS/750 supports the development of reliable programs which are easier to debug and maintain than the corresponding GPSS programs.

However, the basic concepts and internal organization of GPSS have much to recommend itself. In fact, apart from some minor differences in the design and structure of the language, the GPSS/750 system is based entirely on the various entity concepts introduced by GPSS and it mimics the GPSS scheduler.

1.2 Basic Concepts OF GPSS/750

GPSS/750 supports a process-oriented world view of a simulation model which enables the representation of a set of parallel processes which may cooperate with each other and also compete for resources. In order to map this framework onto sequential execution within a uniprocessor environment, GPSS/750 handles time discretely so that the system is examined only at discrete instances of simulated time. The passage of time is modeled by a simulated clock which is used to provide correct sequencing of events in a model. This clock is always incremented to the time of the next scheduled event rather than by a fixed increment. Even though such a strategy requires more sophisticated system software, it is justifiable on the ground that it uses the computer more efficiently.

A simulation model is fully described by its static structure and dynamic behaviour. The static structure of the model is represented by two classes of entities with their attributes :

1. Permanent entities (facility, storage, queue, user chain, logic switch, savevalue, statistical table).
2. Transient entities (transactions).

The dynamic behaviour of the model is represented by a network of instructions logically interconnected to described the life cycles of classes of transactions. GPSS instructions are called BLOCKS and there are three different classes of them : declaration, control and executable blocks. Further discussion on block instructions is deferred until section 2 under 'Language Specifications'.

1.3 Permanent Entities

Although only 3 of the GPSS permanent entities are implemented in GPSS/750, the user will find that they are sufficient for modelling very complex systems.

Associated with every permanent entity is a set of system defined attributes called Standard Numerical Attributes (SNAs); a storage entity has an additional user defined attribute which specifies its maximum capacity whereas a queue entity has one which specifies the type of transactions in the queue.

1.3.1 Facility

This is an entity that can only accept one transaction at a time; it is used to model single server components of a system.

Examples : gasoline pump, computer terminal, bank cashier.

1.3.2 Storage

A storage may be visualised as a pool of one or more facility entities; it can accomodate one or more transactions, depending on its user defined capacity.

Examples : a supermarket parking lot, buffer storage of a computer or a service station.

1.3.3 Queue

This is a statistic-gathering entity that may be useful for gathering statistics on the time that a transaction waits for a service or condition.

Examples : vehicles at a traffic light, jobs waiting to be processed by a computer.

1.4 Transient Entities

Transactions are the units of traffic in a simulation system. They are generated and destroyed explicitly under user program control by entering a GENERATE and a TERMINATE block respectively. After generation, a transaction moves through the model in zero simulated time until blocked by one of the following conditions :

1. Time delay imposed by an ADVANCE block (e.g. the service time in a facility).
2. Refused entry into the next block because of certain logical condition (e.g. trying to seize a facility which is busy).
3. Deactivation in a user chain.
4. Destroyed by a TERMINATE block.

The movement of transactions through the network of block instructions is controlled by the operations of the blocks used. The operations of blocks define the dynamic behaviour of the model by changing the attributes of either permanent or transient entities on which the operations are performed.

The fundamental difference between GPSS and GPSS/750 is in the treatment of transient entities. In GPSS, every transaction has a fixed set of SNAs. Even though the user may specify the number and the type of parameters for a transaction type (e.g. 4 half-word parameters, 10 full-word parameters), the system will always allocate the largest amount of storage (in this case, 10 full-words), which is rather wasteful in terms of storage. The implication here is that there is only one type of transaction in a GPSS system. In order to model more than one logically distinct class of objects in a system, the user would have to map the parameters of the transaction mentally onto different conceptual attributes. For example, parameter 1 of a transaction might be used to store the vertical altitude of a jet fighter, whereas the same parameter of another transaction might well be used to represent something completely different, such as the status of a gasoline pump.

GPSS/750 allows the user to declare and use separate transaction types to represent logically distinct classes of objects. Other than the set of system defined SNAs, each transaction type may include its own distinct set of user defined attributes tailored to suit the needs of the model. This feature of GPSS/750 makes it conceptually superior to the original GPSS system.

1.5 Computational Entities

For modelling purposes, it is frequently necessary to sample from non-uniform distributions (e.g. exponential arrival rate of transactions, service time based on observed data). GPSS/750 allows the user to represent such distributions as FUNCTIONs in the model.

1.5.1 Empirical distributions

Empirical distributions are based on observed data. The data are submitted into the system as a set of points which defines the relationship between the function argument (the independent variable) and the value of the function (the dependent variable). The function argument specifies the source of random numbers (one of eight system provided random number generators) to be used in the sampling process. To make it easier to use the data, GPSS/750 provides two system defined functions for describing empirical distributions : discrete (D) and continuous (C). When a function is referenced, its argument is evaluated and the type of the function determines the function value

returned. If the value of the argument falls between two given points, the discrete function assumes the value of the second point whereas the continuous function performs a linear interpolation to give a uniform distribution between the points.

1.5.2 Theoretical distributions

Several theoretical distributions which are commonly used in queueing models are also implemented in GPSS/750 as system defined distributions. These include the uniform, exponential and normal distributions. Hence, functions used to represent them can be defined very easily by specifying the names of the distributions required.

Example : FUNCTION Arrival, RN6, EXPON, 1;

> Arrival represents the standard exponential distribution (mean=1).

1.6 Sequencing Of Events

As mentioned in section 1.4, the movement of a transaction through a network of BLOCKS alters the system state. This change of state is an event. Thus by sequencing the movements of transactions we are implicitly scheduling the occurrences of events in the model.

To control the logical sequence of simulation GPSS/750 arranges transactions in two system maintained chains known as the Future-Events Chain (FEC) and the Current-Events Chain (CEC). The Block Departure Time (BDT) attribute of a transaction determines which chain it is on; BDT is the time when the transaction is scheduled to move if not prevented by a logical condition. At current clock time = C, the contents of the 2 chains are as follow :

1. CEC - a list of transactions with BDT = C.
2. FEC - a list of transactions with BDT > C.

The contents of CEC in GPSS are slightly different from that of this system in that the GPSS CEC contains transactions with BDT <= C. This is the result of two very different approaches used by the systems to handle transactions which are blocked by a status condition.

In GPSS, a blocked transaction is returned to the CEC and remains there until the condition it is waiting for occurs. Thus when the system clock is advanced again, all those blocked transactions will have their BDT < C', where C' is the new clock time. Whenever the state of the system is changed by another transaction moving through the system (e.g. by releasing a facility or leaving a storage), the whole CEC has to be rescanned to see if the current state of the system allows for any of the previously blocked

transactions to continue its journey through the model again.

GPSS/750 recognizes the inefficiency of having to rescan the CEC everytime the system state is altered. As a solution to this problem, GPSS/750 maintains a separate chain for every storage and facility in the model. When a transaction is blocked while trying to gain control over one of those entities, it will be chained into the local chain associated with the entity. Thus, instead of rescanning the CEC for every status change which occurs in the system, a local chain needs only be scanned when the status of its owner is changed. Furthermore, because the chains are maintained in a First-In-First-Out (FIFO) order within each priority class, the re-assigning of the entity to the first transaction in its chain is simple and straight forward.

Before a simulation run begins, one transaction from each GENERATE block is implicitly created and placed into the FEC. The NEXT_BLK attribute of the transaction is set to the block number of the respective GENERATE block. Conceptually speaking, these transactions are not yet in the system, their creation times are specified by their BDTs. When any one of these transactions is scheduled to move, it will enter its GENERATE block thus signifies its creation. The creation of the current transaction also triggers the creation of a new transaction in the future, i.e. bootstrapping.

During simulation, because events must be executed in chronological order, the FEC and CEC are maintained in ascending order of BDT, with the transaction having the minimum BDT at the head of the chain. In models where different transaction priorities are used, the CEC has to perform an additional task of grouping transactions into separate priority classes. In GPSS/750 this is done by maintaining separate chains for different priority classes. The detailed structures of FEC and CEC are presented in section 1.9 and 1.10 respectively.

To execute events, the GPSS/750 scheduler moves transactions in the following manner :

1. The CEC is scanned from the higher priority chains to lower priority chains. The first transaction within the current priority class is taken out of the chain and moved as far as possible until it is blocked by one of the conditions specified in section 1.4.
2. The above process is repeated until all the chains in CEC are empty.
3. The system clock is updated to the time of the next imminent event and FEC is scanned so that all transactions with BDT = new clock time are transfered over to their respective chains in

CEC according to their priority levels.

4. Steps 1 --> 3 are repeated until one of the following conditions arises :

   a. Both CEC and FEC are empty.
   b. User defined total simulation time is exceeded.
   c. The total number of transactions flowing through the model has exceeded a user defined limit.

It is this multi-level repetitive operation of the event scheduler, necessitated by the requirement to map a set of parallel processes into a single execution sequence, which accounts for the high resource requirements of the two systems. The situation is comparatively worse in GPSS because of the rescanning required for the CEC.

1.7 Executing Events

GPSS/750 retains an image of all the block definition statements used by a model in a PASCAL CASE statement. When a block is defined, it is assigned an unique block number to be used for selection in the CASE statement. The NEXT_BLK attribute of a transaction determines the next operation to be performed.

Example:

```
GENERATE( -,-,-,-,-,- );
QUEUE( q1 );

% BEGIN
    i := diskarm;
    IF i=1 THEN SEIZE( fac1 )
    ELSE SEIZE( fac2 );
  END; %

  ADVANCE( 20 );

% IF I=1 THEN RELEASE( fac1 )
  ELSE RELEASE( fac2 ); %

  DEPART( q1 );
  TERMINATE( 1 );
       .
       .
```

The GPSS/750 preprocessor transforms the above definition statements into the following CASE statement :

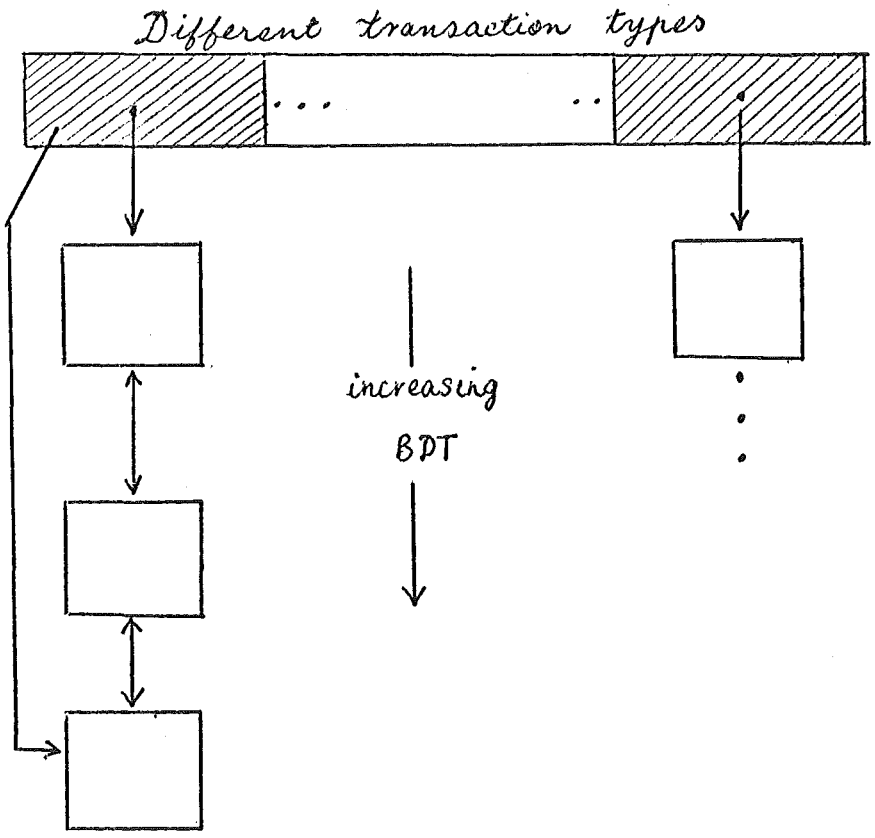
```
CASE NEXT_BLK OF

1 :   GENERATE( -,-,-,-,-,- );
2 :   QUEUE( q1 );
3 :   BEGIN
        i := diskarm;
        IF i=1 THEN SEIZE( fac1 )
        ELSE SEIZE( fac2 );
      END;
4 :   ADVANCE( 20 );
5 :   IF i=1 THEN RELEASE( fac1 )
      ELSE RELEASE( fac2 );
6 :   DEPART( q1 );
7 :   TERMINATE( 1 );
            .
            .
            .
```

The above example also demonstrates how PASCAL compound and single statements may be included and be treated exactly as any GPSS/750 statement.

1.8 Entity Representations

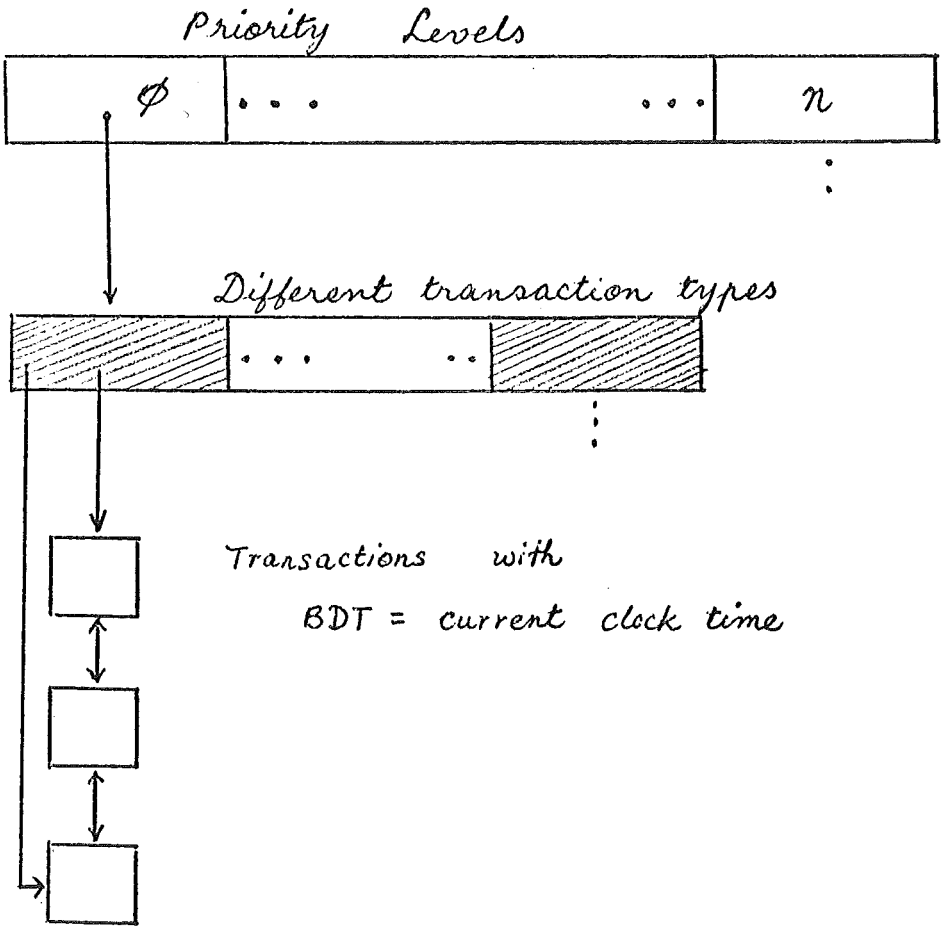
At this point, it is necessary to point out that PASCAL record structures are used to represent different entity types in GPSS/750. The SNAs are implemented as record fields. In a model where more than one transaction type is involved, the sequencing and execution of events becomes a very time consuming and complicated task because the event scheduler is forced to treat transactions of different types separately. This is due to the fact that PASCAL requires the use of separate pointer types for referencing different record structures. Consequently, it is very awkward to link different transaction types into a common list structure. The organization of GPSS/750 chain structures are therefore very different from their counterparts in GPSS. Specifically, we will discuss the organizations of the FEC and CEC in the next two sections. Local chains of server entities are organized exactly like the CEC.

1.9 Organization Of The Future-Events Chain



To find the time of the next imminent event(s), the scheduler has to scan all the chains to find the minimum time among them. To simplify this scanning process, the BDT of the first transaction in each chain is stored in the chain head. Each chain head also contains an attribute specifying the type of transactions in the chain so that appropriate pointers may be used to accessed its contents.

1.10 Organization Of The Current-Events Chain



Transactions are grouped into different priority classes. Owing to the overheads incurred in scanning this multi-level structure, the number of priority classes and the number of transaction types in the system should be kept to a minimum. The current GPSS/750 implementation limits the total number of priority classes to 6 (0..5), with a default value of 0.

Within each priority class the structure is the same as the FEC, with one separate chain per transaction type.

Since all the transactions in CEC are due to move at the current simulated time, their orders in the chains are not important. Nevertheless, they are maintained in a FIFO manner.

2 GPSS/750 LANGUAGE SPECIFICATIONS

One of the advantage of GPSS is that it can be used effectively even if only a subset of instructions is known. A workable subset of GPSS, sufficiently complete to solve practical problems, have been defined for GPSS/750. This subset consists of :

1. 5 entity definition statements.
2. 13 block definition statements.
3. 5 control statements.

A GPSS/750 program consists of a collection of mainly these 3 classes of statements grouped together in the above specified order. The syntax of these statements follows closely to that of the GPSS system, with some minor variations to suit the new system. Apart from the restriction imposed upon the order of which statements of different classes may appear in a program, the general layout of the program (e.g. which column should a statement begins, how many statements per line) is left entirely to the programmer. This is a deviation from the 'one instruction per line' philosophy adopted by the GPSS system. It should also be noted that GPSS/750 uses semicolon, ';', as a statement terminator. Every statement must be terminated by a semicolon; the only exception is the 'END.' control statement which is terminated by a period.

In the sequel of this section, GPSS/750 reserved words are typed in bold letters. They have special, fixed meanings in GPSS/750, implied by their occurrence in the syntax of the language. A list of all GPSS/750 reserved words is given in Appendix A.

2.1 Entity Definition Statements

Every entity used in a GPSS/750 program must be declared in a entity definition statement before it is referenced. This is very different from the GPSS system which requires declarations only for those entities which have user defined attributes (e.g. storage, computational entity, savevalue, statistical table). Other entities are implicitly created by the system when they are first referenced in the program. This difference in design is a direct consequence of the type checking facility provided by the GPSS/750 system.

After their declarations, the system automatically initializes the SNAs of all the entities to their respective 'initial' values (e.g. chains are initialized to be empty, statistical counters are initialized to zero).

2.1.1 TRANSACTION

There are 2 types of transaction definition statements :

1. Define transactions with only the standard set of system defined attributes.

   ```
   TRANSACTION A ;
   ```

   Operand :

   A : transaction name (a valid identifier).

2. Define transactions with extra user defined attributes.

   ```
   TRANSACTION A = A1 : T1 ;
                   A2 : T2 ;
                   .
                   .
                   An : Tn ;
                END ;
   ```

   Operand :

   A : transaction name (a valid identifier).
   Ai (i = 1..n) : user defined attributes.
   Ti (i = 1..n) : PASCAL type identifiers.

The current GPSS/750 implementation does not prevent the user to declare two or more transaction types which are structurally identical to each other, as in the following example :

```
TRANSACTION A ;
TRANSACTION B ;
TRANSACTION C ;
```

It is the responsibility of the programmer to minimize the number of unnecessary transaction types within a system inorder to improve the efficiency of the simulation run.

2.1.2 Storage A Capacity B ;

This declaration statement defines a storage entity and its capacity.

Operands :

A : Storage name (a valid identifier).
B : Unit of capacity defined for the storage (integer number).

2.1.3 Facility A ;

This is a declaration statement for a facility entity.

Operand :

A : Facility name (a valid identifier).

2.1.4 Queue A OF B ;

This statement defines a queue entity and the type of its members.

Operands :

A : Queue name (a valid identifier).
B : Specifies the type of transactions allowed into the queue. This operand can therefore be used as an additional security check against transactions inadvertantly entering a wrong queue. Alternatively, the reserved word 'ALL' may be specified to give entry to all transactions in the system.

2.1.5 Function A, B, C, D ;

This declaration statement defines a GPSS/750 function.

Operands :

A : Function name (a valid identifier).
B : The function argument - can be any one of the eight system provided random number generators (RN1 --> RN8).
C : Function type - one of :

a. UNIFORM - uniform distribution.
b. EXPON - negative exponential distribution.
c. NORMAL - cumulative normal distribution.
d. D - discrete function.
e. C - continuous function.

D : Function or distribution attribute -
a. range of values for UNIFORM distribution in the form x-y.
b. non-zero mean value for EXPON distribution.

c. mean and standard deviation of NORMAL distribution in the form x,y.

d. total number of point pairs for D and C functions. The point pairs must be given immediately following the function definition statement in the following format :

   /x1,y1/x2,y2/...../xn,yn/ , where xi,yi are either integer or real numbers.

   The given pairs of numbers will be read into two system defined arrays : <function name>_XVAL and <function name>_YVAL. <function name> stands for the name of the function declared in the function definition statement.

   This is obviously very tedious and impractical especially when a large number of values (e.g. 1000 pairs) are needed, as in many large simulation models. GPSS/750 provides the user with the option of reading data into the two system generated arrays by using simple PASCAL READ statements. This can be done by specifying a special 'no data' symbol, '///' (two slashes without any blank between them), after the funtion definition statement. The user may then include his own input module any where in the program for reading external data into the arrays before the simulation run.

2.2 Block Definition Statements

The 13 block definition statements are implemented as procedure calls, with parameters enclosed in a pair of left and right parentheses.. The parentheses are coerced into the syntax of the GPSS block instructions to improve the readability of the program.

In the following discussion, where a default value is specified for an operand, the operand will assume the default value if it is omitted. Otherwise all the operands must be supplied by the user.

The block instructions are grouped according to the nature of their operations.

2.2.1 Transaction-oriented blocks

GENERATE(A, B, C, D, E, F) ;

Operation :

This block creates transactions that enter the model in the next sequential block. During a simulation run, a run time error will occur if a transaction attempts to enter this block from any other blocks.

Operands :

A : Name of transaction type.
B : Generation time of the first transaction.
C : Mean intergeneration time (integer or real number); may be a function name.
D : One of -

a. the spread time - (integer or real number) intergeneration time is uniformly distributed in the interval (C-D, C+D). D must be less than C.
b. function modifier - intergeneration time is given by the mean multiplied by the value returned by calling the function specified.

E : The priority level (0..5) to be assigned to each of the transaction created.
F : The maximum number of transactions to be generated by this block (integer number).

Example :

GENERATE(Request,0,10,5.5,4,1000) ;

- generates transactions of the type Request at intervals uniformly distributed between 4.5 and 15.5 model time units; the first transaction is generated at time 0.

- priority level of transactions is 4.

- 1000 transactions are to be generated by this block.

TERMINATE(A) ;

Operation :

This block destroys the entering transaction, thus removing it from the system.

Operand :

A : The number of units by which the termination count is reduced (integer number). If this is not specified, the termination count is unaltered.

ADVANCE(A,B) ;

Operation :

This block provides the means to delay transactions. Transactions entering this block will be delayed for a period of time computed from the A and B operands.

Operands :

A : The mean delay time (integer or real number); may be a function name.
B : Optional, default value is 0. If specified, can be one of :

a. the spread time (integer or real number).
b. function modifier. (a and b are treated exactly the same as in GENERATE block)

Examples :

1. ADVANCE(100,5) ;

   - transaction delayed for a length of time uniformly distributed in the interval 95-105.
2. ADVANCE(Speed,Fluct) ;

   - transaction delayed for a length of time given by the value of function Speed multiplied by the value of function Fluct.

PRIORITY(A) ;

Operation :

This block is used to assigned a new priority level to the entering transaction. The priority level of individual transaction plays an important role in the sequencing of transaction movements through the model. Transactions with higher priority level always have preference over those with lower priority level. Within each priority level, the transactions are treated in a first come first serve basis.

Operand :

A : New priority level to be assigned to the transaction (0..5).

2.2.2 Equipment-oriented blocks

FACILITY

SEIZE(A) ;

Operation :

This block tests the status of the specified facility. If the facility is busy (i.e. currently used by another transaction), the transaction is blocked from entering the block and is kept waiting in the local queue associated with the facility.

If the transaction enters the block, it will immediately seize the facility and proceed to the next block. All subsequent attempts by other transactions to seize the facility will be blocked until the facility is released again.

Operand :

A : Facility identifier.

RELEASE(A) ;

Operation :

When a transaction enters this block, it relinquishes control of the specified facility. A run time error will occur if the transaction does not have the specified facility under its control at the time of entry.

This block also performs the task of re-assigning the newly released facility to a waiting transaction in the local queue of the facility.

Operand :

A : Facility identifier.

PREEMPT(A);

Operation :

If the specified facility entity is not in use, the entering transaction seizes and gain control of it. Otherwise, the transaction will only succeed in preempting the facility if it has a priority level that is at least equal to the current user of the facility. A preempted transaction waits in a separate preempted-queue which has a higher priority than the other wait queue.

Operand :
A : Facility identifier.

RETURN(A) ;

Operation :

The entering transaction relinquishes control of the specified facility. The free facility will be returned to the control of a previously preempted transaction to complete its remaining units of service time.

The preempted-queue is arranged in a Last-In-First-Out order so that the most rescently preempted transaction will regain control of the facility first.

A run time error will occur if the transaction does not have the specified facility under its control on entering this block.

QUEUE

QUEUE(A,B) ;

Operation :

This block restricts the type of transactions entering the queue entity specified by the A-operand. If the transaction type is compatible with the queue type, the transaction enters this block and the queue contents are incremented by a total number of units specified by the B-operand. The transaction then proceeds to the next block as a member of the queue.

A run time error will occur if a transaction attempts to join an incompatible queue.

Operands :

A : Queue identifier.
B : Total units added to the queue contents (integer number); default 1.

DEPART(A,B) ;

Operation :

If the entering transaction does not belong to the member type of the queue, a run time error occurrs. Otherwise, the queue contents are decremented by a number of units specified by the B-operand. A transaction needs not depart with the same number of units with which it entered the queue but it must not be greater than the remaining contents of the queue.

Operands :

A : Queue identifier.
B : Total units decremented (integer number); default 1.

STORAGE

ENTER(A ,B) ;

Operation :

Entry to this block is granted to a transaction if and only if the number of units requested is not greater than the remaining capacity of the specified storage entity. When a transaction enters this block, the capacity of the specified storage is decremented by the number of units specified by the B-operand and the transaction proceeds to the next block.

If the remaining capacity of the specified storage is not enough to accomodate the incoming transaction, the transaction will be blocked and sent to the local queue associated with the storage.

Operands :

A : Storage identifier.
B : Storage units requested by the transaction (integer number); default 1.

LEAVE(A,B) ;

Operation :

The entering transaction relinquishes a specified number of storage units and proceeds to the next block. A transaction needs not leave with the same number of units with which it entered the storage entity but the current contents must be < maximum contents as a result.

Operands :

A : Storage identifier.
B : Total number of storage units to be relinquished (integer number); default 1.

2.2.3 Transaction-flow modifications

When the executions of simulation events were discussed in section 1.7, it was mentioned that the NEXT_BLK attribute of a transaction determines the next operation to be performed. Thus the flow of control in a model can be modified by changing the content of the NEXT_BLK attribute. A parallel can therefore be drawn between the NEXT_BLK attribute of a transaction and the program counter of a conventional program.

In GPSS/750, the transaction flow modification is provided by the TRANSFER block. For this purpose, a label is declared and used to associate the block number of a block with a symbolic name. This is similar to the concept of using symbolic names to address computer memory locations.

TRANSFER(A,B,C) ;

Operation :

The entering transaction is directed to a specified location, depending on the transfer mode defined by the A-operand.

Operands :

A : The transfer mode
  a. omitted - transaction unconditionally sent to the block address specified by the B-operand. In this mode, the C-operand should not be coded.
  b. statistical - the mode is a decimal fraction, such as 0.25, giving the probability that address specified by the C-operand will be chosen. The rest will be sent to the address specified by the B-operand.

B : a label name in the program.
C : a label name in the program.

Examples :

1. TRANSFER(loop) ;
   - entering transactions are unconditionally sent to block location 'loop'.
2. TRANSFER(0.25,Good,Bad) ;
   - 25% of the transactions entering this block are being sent to the location 'Good' while 75% are being sent to location 'Bad'.

2.3 Control Statements

The control statements offer the user some control over a simulation run.

SIMULATE(A) ;

Operation :

This block requests a simulation run if no syntax errors were detected by the preprocessor.

Operand :

A : Processing time limit in minutes.

START(A,B) ;

Operation :

This block initiates the simulation run if no syntax errors were detected by the preprocessor.

Operands :

A : Run termination count.
B : Print suppression field, NP (optional). If this operand is specified, the statistical printout at the end of the current run will be suppressed.

CLEAR ;

This block clears the entire system - all transactions and statistics belonging to the preceding run are destroyed. However, the seeds for the eight random number generators are not reset.

RESET ;

This is a restricted form of CLEAR statement in that only the statistics gathered during the preceding run are erased, the current state of the model is preserved. This feature is useful for deferring the statistics gathering until the model has hopefully reached some equilibrium after some estimated period of model time.

END.

This block marks the end of a GPSS/750 model definition. Other PASCAL constructs may follow this card.

2.4 Miscellaneous Language Features

So far in this section, we have used the term 'valid identifier' in various places where a user defined identifier is required in the syntax of the language. A valid identifier in GPSS/750 is defined as 'a sequence of characters beginning with a letter and followed by zero or more letters or decimal digits'. The maximum length of an identifier should not be more than 72 characters (length of one input line), and the first 11 characters must be unique.

The GPSS/750 preprocessor allows lower and upper case letters to be used interchangeably to improve the readability of the program. No special characters are allowed (e.g. '_', '-') to form an identifier. This restriction is necessary to avoid name conflicts between user defined and preprocessor generated identifiers. In places where name conflicts can occur, all preprocessor generated identifiers are generated with a under_bar character, '_'; identifiers not likely to cause any problems are those qualifiable by a record structure (i.e. a record field), local identifiers used in system procedures or functions, and GPSS/750 reserved words.

As mentioned at the beginning of this report, PASCAL statements may be embedded in a GPSS/750 program. They are ignored by the preprocessor but will be included in the generated PASCAL program to be compiled by the PASCAL compiler. A pair of '%'s must be used to enclose a group of such statements and any number of such 'alien' groups may be included in various parts of the program.

Comments in GPSS/750 are enclosed between special brackets '(*' and '*)'. It may cover more than one line and may be placed anywhere in the program text, except in the middle of symbols such as an identifier or a number. Blank lines may also be used to improve the readability of the program.

2.5 Addressing Mode

The only way to specify a GPSS/750 entity is by giving its name directly as in most programming languages. It would not be too difficult to implement the GPSS indirect addressing feature, all it needs is a binary tree for storing all the entities and their entity numbers. Where the entity number is given instead of the entity name, the name could be obtained by searching the tree. However, it was decided not to include this rather powerful concept of GPSS in the current implementation because of the time constrain.

The following example, however, demonstrates how a PASCAL compound statement can be used to solve a problem involving indirect addressing.

Example :

```
SEIZE FN$diskarm
LEAVE Q
   .
   .
```

Transactions entering the above GPSS block will seize one of a range of facility entities depending on the value returned by the function called diskarm. The solution in GPSS/750 would be :

```
      .
      .
% BEGIN
     i:=diskarm;
     CASE i of
      1: SEIZE( facility1 );
      2: SEIZE( facility2 );
      .
      .
      END (* CASE *)
  END;
 %
  LEAVE( Q );
      .
      .
```

3 GPSS/750 IMPLEMENTATION

The implementation of GPSS/750 is based on a preprocessor system. To run a simulation program, the user submitts his program to the preprocessor to be checked for syntax errors. As the preprocessor parses the input program, it also builds up a PASCAL program and a listing of the source program. If no syntax errors were detected by the preprocessor, the resultant PASCAL program is submitted to the PASCAL compiler for compiling before the actual simulation run. However, if syntax errors were detected during the precompilation stage, the programmer would have to resubmit a corrected version of his program until it is rid of all syntax errors.

The printing of the PASCAL output program is suppressed as soon as the preprocessor detects an error in the input program. However, the preprocessor continues to parse the rest of the program, inserting appropriate error messages in the listing for diagnostic purposes at the same time.

The preprocessor is composed of three main modules :

1. Error recovery routine.
2. Scanner.
3. Parser.

In the following sections, the basic functions of the modules are discussed briefly. It is assumed that the reader of this report has a knowledge of preprocessing and compiling techniques. For detailed structure of the preprocessor, the reader is referred to the source program listing submitted with this report.

3.1 Error Recovery Routine

In the event of an error, many unwanted error messages might be created because the parser is 'confused by unexpected symbols'. The concept of followset is used to solve this problem.

The followset consists of a set of symbols which are expected by the parser if an error has not occurred. Each of these symbols denotes the start of a new operation regardless of whether an error has occurred. For instance, the keyword 'TRANSACTION' signals the start of a transaction definition. After an error has occurred, one or more

symbols will be skipped until one of the followset symbols is encountered.

3.2 Scanner

The Scanner is a specialized module whose task is to obtain the next symbol from the input program. It reads one line of text at a time and stores it into a character array. When invoked, it assembles the next string of characters and classifies it as one of : reserved word, special character, integer number, real number, or an ordinary identifier. The information returned by the scanner is used by the parser for checking the syntax of GPSS/750 programs.

3.3 The Parser

This is the main workhouse of the preprocessor where the syntax of GPSS/750 programs are being checked. In order to perform this task correctly, the parser has to keep track of all the identifiers which are declared to be entity names or labels in the model.

All the identifiers declared in the program are stored as separate nodes in an unbalanced binary tree. There are six different types of nodes and each node consists of a fixed part and a variable part which depends on the type of entity declared.

The fixed part consists of :

1. Name - user defined entity name.
2. Llink - pointer to the left node.
3. Rlink - pointer to the right node.
4. Next - pointer to the next node
   of the same entity type.

The variable part of different node types consists of :

Queue

1. Xtyp - a pointer to the member transaction type; if the queue is declared to be of type 'ALL', i.e. no restriction on member type, this pointer is set to NIL.

Storage

1. Stor_cap - user defined storage capacity.

Function

1. Ftyp - function type, e.g. NORMAL, UNIFORM.
2. Tot - total number of point pairs for a discrete or continuous function.

Label

1. Lval - the value assigned to the label.
2. Defined - a boolean flag indicating whether a label has been defined, i.e. assigned a value.

Facility and Transaction have no variable parts.

The various fields of the nodes are used to store information which will be required for generating correct PASCAL codes for the entities declared.

A new node is created for every new identifier declared in the program. The position of the node in the tree is determined by comparing the new identifier with the nodes of the tree in the following manner, starting from the root of the tree :

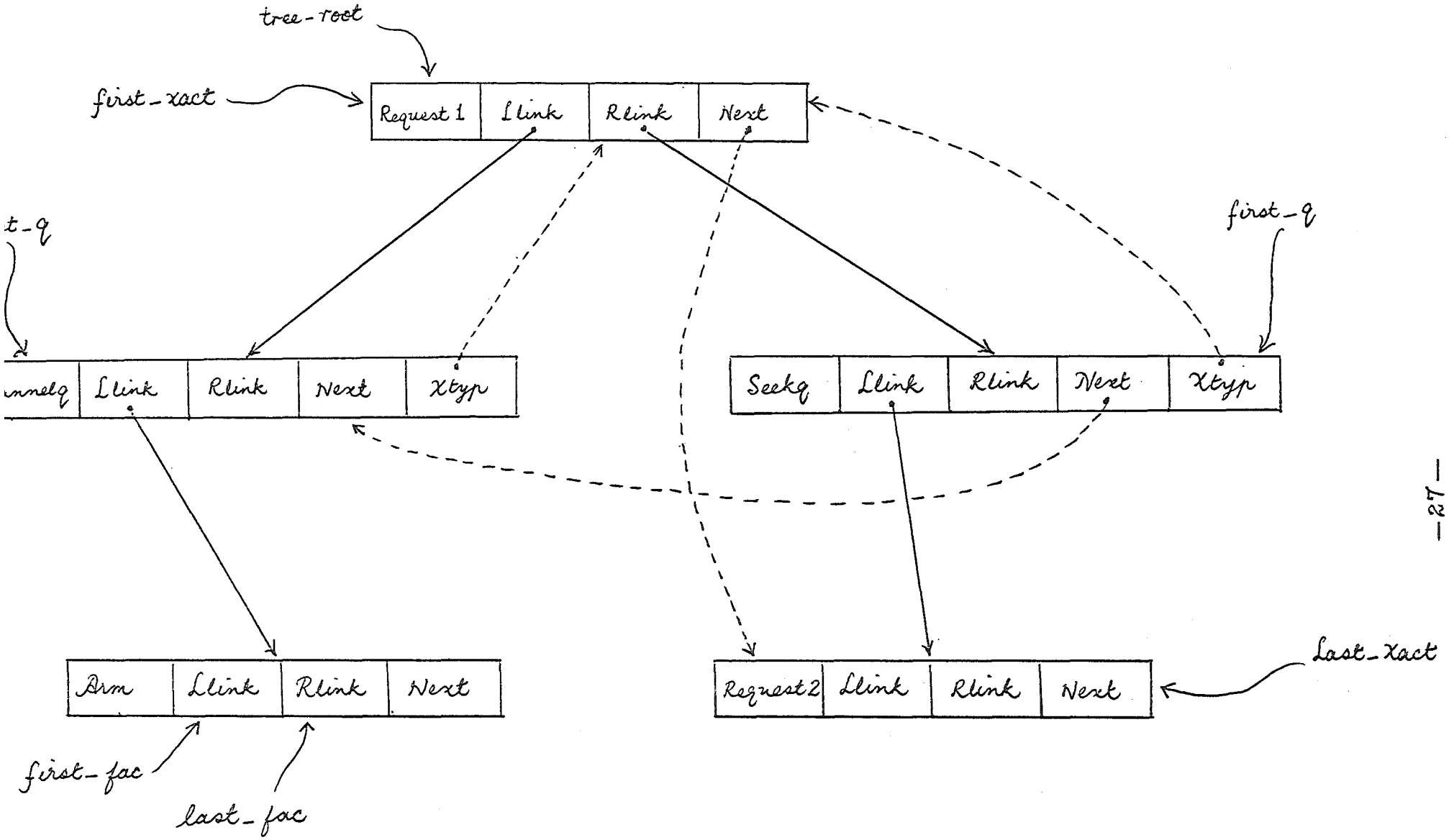
1. If the node we are currently looking at is alphabetically greater than the new node, follow the left link to the next node, otherwise follow the right link; an error message, 'identifier declared twice', will be invoked if the new node is alphabetically equal to the node we are looking at. The offending identifier will still be stored regardlessly so that its use later in the model will not be flagged as an undeclared entity.
2. Repeat step 1 until the link concerned is NIL.
3. Append the new node to the link.

As statements such as 'FOR CHAIN:=first transaction type TO last transaction type DO ....' have to be generated very frequently, two more pointers are maintained for every node type, e.g. FIRST_XACT points at the first transaction node in the tree and LAST_XACT points at the last transaction node. The NEXT link of each node is used to link all the nodes of the same type in a linked list structure. In this way, when searching for an identifier of a particular type (e.g. queue), it is more efficient to follow the NEXT link especially when the number of entities of a particular type is usually very small.

The following diagram shows the tree after the five entity definition statements have been declared :

```
TRANSACTION request1;
QUEUE seekq OF request1;
TRANSACTION request2=seektime:REAL;
                     wait:REAL;
                     END;
QUEUE channelq OF request2;
FACILITY arm;
```

Preprocessor Tree Structure



For every GENERATE block, the parser also keeps a record of the transaction type, block number, and the generation time of the first transaction. This is required for creating the first transaction of every GENERATE block in the system.

3.4 Current State Of Implementation

The initial design of GPSS/750 was a 'single transaction type' system, with only one structurally distinct class of transaction in the system. The implementation of such a system would be very straight forward as all the block instructions could be implemented as standard system procedures which may be included and used by any program; the four entity types (transaction, queue, storage, facility) could be implemented as predefined entity types, with a set of standard attributes.

A successfully implemented GPSS/750 system will be used for teaching purposes (as opposed to GPSS which was geared towards pratical applications). It was therefore decided that GPSS/750 should allow the representation of logically distinct classes of objects as different transaction types in the system. From the conceptual point of view, this presents no special difficulties. However, from the implementation point of view, it necessitated a redesign of the GPSS/750 system. The major problem in this later design arises from two conflicting requirements :

1. The system is required to simulate the interactions between different transaction types within a model.
2. Owing to the underlying structural differences between different transaction types, PASCAL requires them to be treated separately.

A major part of this project was involved with the design of the GPSS/750 system, which in itself is a very substantial undertaking. The implementation of the proposed system has been started but it is still at its testing stage. The implementation to date is more concerned with the basic concepts of GPSS/750 rather than a full implementation of the language. Once the basic concepts have been successfully implemented, any further extensions to the language could be made with a minimum of effort.

A few PRIME system features have been incorporated into the current implementation. One of these is the use of under_bar characters in preprocessor generated identifier names mentioned earlier in section 2.4. GPSS/750 also makes use of a PRIME system procedure, CTIM$A, for calculating the total processor time used by a simulation run. Finally, the DISPOSE procedure implemented on PRIME is a feature of standard PASCAL which is not implemented on many PASCAL installations.

3.5 Suggestions For Improvement

Only the basic concepts and a small subset of GPSS block instructions have been included in the current version of GPSS/750. Even though its present form, if successfully implemented, can be used effectively to solve very complex problems, future versions could be extended progressively by adding new instructions and by introducing some other GPSS concepts (e.g. indirect addressing) into the system.

The optimization of codes generated by the preprocessor is another aspect of implementation which is worth looking at. At the moment, all the codes generated by the preprocessor are designed to handle more than one transaction types. Such a system requires the use of many CASE and IF statements to verify the type of transaction currently active in the model. All these testings are obviously unnecessary when there is only one transaction type in the system.

CONCLUSIONS

The purpose of this project has been to analyse the GPSS simulation language and to reveal the problems associated with its design and implementation. As a result of this analysis, a new system which includes a subset of the GPSS language has been designed for the PRIME computer system.

Acknowledging that a good implementation requires a solid foundation provided by a sound system design, the emphasis of this project has been the design of the proposed GPSS/750 system.

A complete implementation of the proposed system has not been possible within the time constrain of this project because of the large amount of time spent on various designing aspects of the system. A listing of the preprocessor source program with a few test runs of the program are submitted with this report. The program and the system as a whole have not been formally documented, its submission with this report is only intented to give the reader some idea on how such a system could be implemented. It is hoped that the findings of this project would be benefitial to the future implementor of similar systems.

APPENDIX A

GPSS/750 RESERVED WORDS AND SPECIAL SYMBOLS

Reserved Words

A reserved word is a word which has a special meaning and cannot therefore be used as an identifier. The following is a complete list of GPSS/750 reserved words :

C	D	OF
NP	ALL	BTU
END	RN1	RN2
RN3	RN4	RN5
RN6	RN7	RN8
BLOCK	CLEAR	CLOCK
ENTER	EXPON	LABEL
LEAVE	MODEL	QUEUE
RESET	SEIZE	START
TOTAL	ASSIGN	DEPART
NORMAL	RETURN	ADVANCE
MAXIMUM	PREEMPT	RELEASE
STORAGE	UNIFORM	CAPACITY
FACILITY	FUNCTION	GENERATE
PRIORITY	TRANSFER	SIMULATE
BEHAVIOUR	TERMINATE	TRANSACTION

Special Symbols

- Comments in a GPSS/750 program are enclosed between '(*' and '*)'.

- '///' is a 'no data' symbol to allow the programmer to read external data into the system.

APPENDIX B

LAYOUT OF A GPSS/750 MODEL

MODEL <model name>;
BTU = <basic time unit>;

Optional Global Constants

- MAXIMUM PRIORITY = <maximum priority level>;
- TOTAL BLOCK = <tot number of blocks used>;

Entity Definition Statements

- TRANSACTION
- STORAGE
- QUEUE
- FACILITY

Label Declarations

BEHAVIOUR

Block Definition Statements

GENERATE
.
.
.
TERMINATE

GENERATE
.
.
.
TERMINATE

One or more transaction flow templates, representing one or more "parallel" processes.

Control Statements

- START
- RESET
- CLEAR

MODEL END.

APPENDIX C

A SAMPLE GPSS/750 PROGRAM

```
 MODEL example;
(**************)
BTU = 1 ms;
MAXIMUM PRIORITY = 0;

(* Parts to be processed by a machine *)
TRANSACTION parts = pname:ARRAY[1..6] OF CHAR;
                    pcode:INTEGER;
                    END;
(* An oven which can bake 2 parts at a time *)
STORAGE oven CAPACITY 2;

(* The machine which processes the parts *)
FACILITY machine;

(* A queue where parts are waiting to be processed *)
QUEUE wait OF parts;


BEHAVIOUR
SIMULATE( 5 );       (* simulate for no more than
                        5 minutes of cpu time      *)

(* life cycle of  parts *)

(* The first part arrives at 0 simulated time.
   - Inter-arrival time is uniformly distributed
     between 2 and 4 time units.
   - priority assigned to each part = 0.
   - a total of 1000 parts is to enter the system.  *)

  GENERATE( parts,0,3,1,0,1000 );

  QUEUE( wait );          (* join the waiting queue      *)
  SEIZE( machine );       (* being processed
                             when the machine is free   *)
  DEPART( wait );         (* leave the queue to be
                             processed by the machine   *)
  ADVANCE( 4 );           (* processing time             *)
  ENTER( oven );          (* if there is still space in
                             the oven, bake it          *)
  RELEASE( machine );     (* the machine is free to
                             process another part       *)
  ADVANCE( 9 );           (* baking time                 *)
  LEAVE( oven );          (* leave the oven              *)
  TERMINATE( 2 );         (* leave the system and
                             decrement the counter by 2 *)
```

```
(* simulation controls *)
  START( 350,NP );      (* start the simulation and
                           stop it after 350 transactions
                           have gone through.
                         - Don't print report          *)
  RESET;                (* erase all the statistics
                           without changing the current
                           system state.                 *)
  START( 1000 );        (* run for 1000 transactions
                           and print the final report *)

MODEL END.
```

REFERENCES

1. Bobillier, P. A. /Kahan, B. C. /Probst, A. R.
   'Simulation With GPSS And GPSS V'
   Englewood Cliffs 1976

2. Bryant, R. M.
   'SIMPAS : A Simulation Language Based On PASCAL'
   Technical Report #55 (Jan. 1981)
   Academic Computing Centre
   The University Of Wisconsin

3. Uyeno, D. /Vaessen, W. /Solecki, A. /Phillips, D.
   'The PASSIM System. PASCAL-Based Simulation Using GPSS Concepts'
   Department Of Business Administration
   University Of British Columbia
   Vancouver, Canada.